Module 9
Search and Sort Algorithms

Tamer Abdou, PhD

IND 830 - Python Programming for Data Science

Data Science Laboratory

Ryerson University

Ryerson University
The Chang School of Continuing Education

Outline

1. Search Algorithms
2. Sort Algorithms

Ryerson University
The Chang School of Continuing Education

Search Algorithms

1 Search Algorithms
Linear Search
Binary Search

2 Sort Algorithms

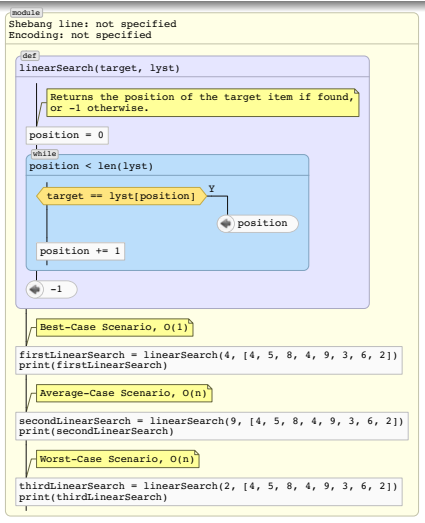
Linear Search: Algorithm Design

- Two inputs are needed: an item and a list
- A loop is constructed to iterate through all the items inside the list
- On each pass, the given item is compared to the current item in the list
- The output is either the position of the matched item or -1 if the item has no match

Example 9.1 (Script)

```
1  def linearSearch(target, lyst):
2      #Returns the position of the target item if found,or -1 otherwise.
3      position = 0
4      while position < len(lyst):
5          if target == lyst[position]:
6              return position
7          position += 1
8      return -1
9
10 #Best-Case Scenario, O(1)
11 firstLinearSearch = linearSearch(4, [4, 5, 8, 4, 9, 3, 6, 2])
12 print(firstLinearSearch)
13 #Average-Case Scenario, O(n)
14 secondLinearSearch = linearSearch(9, [4, 5, 8, 4, 9, 3, 6, 2])
15 print(secondLinearSearch)
16 #Worst-Case Scenario, O(n)
17 thirdLinearSearch = linearSearch(2, [4, 5, 8, 4, 9, 3, 6, 2])
18 print(thirdLinearSearch)
```

Example 9.1 (Control Flow)



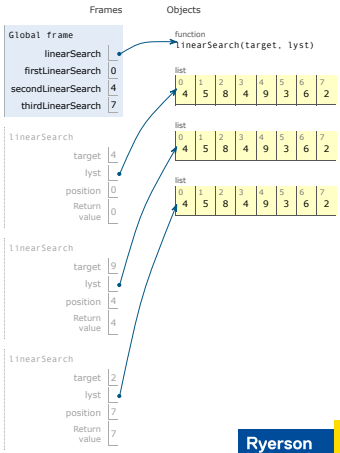
Example 9.1 (Program Execution)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  def linearSearch(target, lyst):
2      #Returns the position of the target item if found,or -1 otherw
3      position = 0
4      while position < len(lyst):
5          if target == lyst[position]:
6              return position
7          position += 1
8      return -1
9
10 #Best-Case Scenario, O(1)
11 firstLinearSearch = linearSearch(4, [4, 5, 8, 4, 9, 3, 6, 2])
12 print(firstLinearSearch)
13 #Average-Case Scenario, O(n)
14 secondLinearSearch = linearSearch(9, [4, 5, 8, 4, 9, 3, 6, 2])
15 print(secondLinearSearch)
16 #Worst-Case Scenario, O(n)
17 thirdLinearSearch = linearSearch(2, [4, 5, 8, 4, 9, 3, 6, 2])
18 print(thirdLinearSearch)
```

Print output (drag lower right corner to resize)

```
0
4
7
```



Ryerson University
The Chang School of Continuing Education

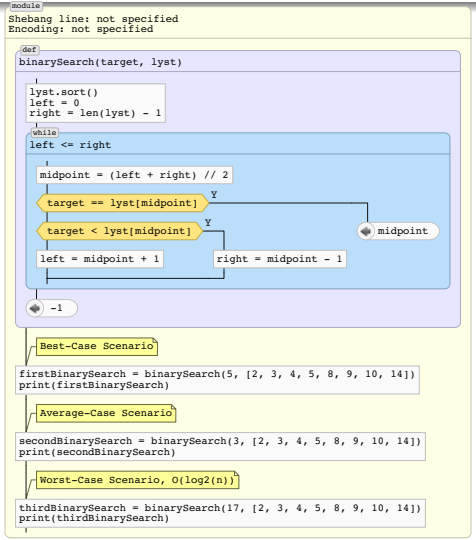
Binary Search: Algorithm Design

- Two inputs are needed: an item and a list
- The items in the list should be sorted in an ascending order
- The given item (target) is compared to the middle position item in the list
- Three-way selection statement:
  1. The target matches the middle position item: Return the position
  2. The target is less than the middle position item: BinarySearch the portion of the list before the middle position
  3. The target is greater than the middle position item: BinarySearch the portion of the list after the middle position
- The algorithm stops when the target is found

Example 9.2 (Script)

```
1   def binarySearch(target, lyst):
2       lyst.sort()
3       left = 0
4       right = len(lyst) - 1
5       while left <= right:
6           midpoint = (left + right) // 2
7           if target == lyst[midpoint]:
8               return midpoint
9           elif target < lyst[midpoint]:
10              right = midpoint - 1
11          else:
12              left = midpoint + 1
13      return -1
14
15  #Best-Case Scenario
16  firstBinarySearch = binarySearch(5, [2, 3, 4, 5, 8, 9, 10, 14])
17  print(firstBinarySearch)
18
19  #Average-Case Scenario
20  secondBinarySearch = binarySearch(3, [2, 3, 4, 5, 8, 9, 10, 14])
21  print(secondBinarySearch)
22
23  #Worst-Case Scenario, O(log2(n))
24  thirdBinarySearch = binarySearch(17, [2, 3, 4, 5, 8, 9, 10, 14])
25  print(thirdBinarySearch)
```

Example 9.2 (Control Flow)



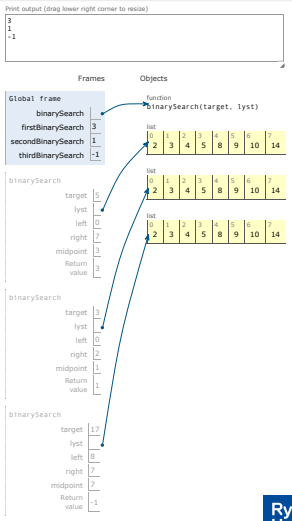
Example 9.2 (Program Execution)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  def binarySearch(target, lyst):
2      lyst.sort()
3      left = 0
4      right = len(lyst) - 1
5      while left <= right:
6          midpoint = (left + right) // 2
7          if target == lyst[midpoint]:
8              return midpoint
9          elif target < lyst[midpoint]:
10             right = midpoint - 1
11         else:
12             left = midpoint + 1
13     return -1
14
15 #Best-Case Scenario, O(1)
16 firstBinarySearch = binarySearch(5, [2, 3, 4, 5, 8, 9, 10, 14])
17 print(firstBinarySearch)
18
19 #Average-Case Scenario, O(log2(n))
20 secondBinarySearch = binarySearch(3, [2, 3, 4, 5, 8, 9, 10, 14])
21 print(secondBinarySearch)
22
23 #Worst-Case Scenario, O(log2(n))
24 thirdBinarySearch = binarySearch(17, [2, 3, 4, 5, 8, 9, 10, 14])
25 print(thirdBinarySearch)
```

Print output (drag lower right corner to resize)

```
3
1
-1
```



Ryerson University
The Chang School of Continuing Education

Sort Algorithms

1 Search Algorithms

2 Sort Algorithms
- Selection Sort
- Bubble Sort
- Insertion Sort

Selection Sort: Algorithm Design

- Only one input is required; either an ordered or unordered list
- A nested loop is required:

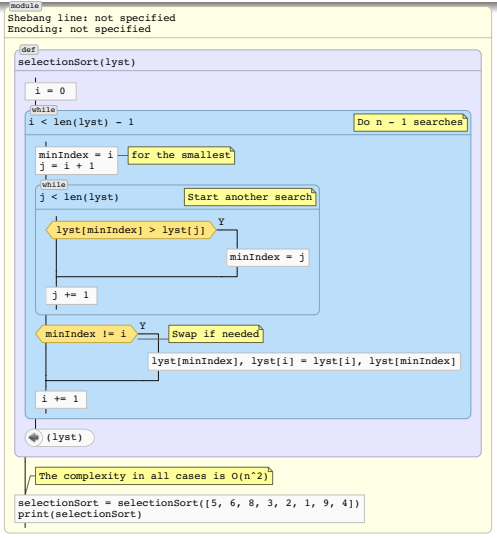
  Outer loop: Traverses the list from the beginning, and on each pass, reorders two elements if needed

  Inner loop: Compares the value of the engaged element with the values of the remaining elements in the list, and returns the position of the element with the least value

Example 9.3 (Script)

```
1  def selectionSort(lyst):
2      i = 0
3      while i < len(lyst) - 1:          # Do n - 1 searches
4          minIndex = i                  # for the smallest
5          j = i + 1
6          while j < len(lyst):          # Start another search
7              if lyst[minIndex] > lyst[j]:
8                  minIndex = j
9              j += 1
10         if minIndex != i:             # Swap if needed
11             lyst[minIndex], lyst[i] = lyst[i], lyst[minIndex]
12         i += 1
13     return(lyst)
14
15 #The complexity in all cases is O(n^2)
16 selectionSort = selectionSort([5, 6, 8, 3, 2, 1, 9, 4])
17 print(selectionSort)
```

Example 9.3 (Control Flow)



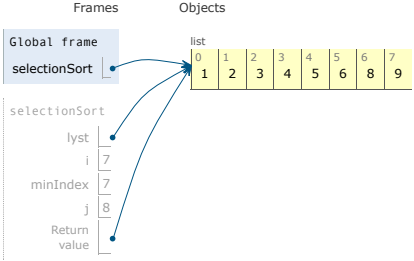
Example 9.3 (Program Execution)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  def selectionSort(lyst):
2      i = 0
3      while i < len(lyst) - 1:          # Do n - 1 searches
4          minIndex = i                  # for the smallest
5          j = i + 1
6          while j < len(lyst):          # Start another search
7              if lyst[minIndex] > lyst[j]:
8                  minIndex = j
9              j += 1
10         if minIndex != i:             # Swap if needed
11             lyst[minIndex], lyst[i] = lyst[i], lyst[minIndex]
12         i += 1
13     return(lyst)
14
15 #The complexity in all cases is O(n^2)
16 selectionSort = selectionSort([5, 6, 8, 3, 2, 1, 9, 4])
17 print(selectionSort)
```

Print output (drag lower right corner to resize)

```
[1, 2, 3, 4, 5, 6, 8, 9]
```



Bubble Sort: Algorithm Design

- One input is needed; either an ordered or unordered list, and on each pass it leaves
- A nested loop is required:

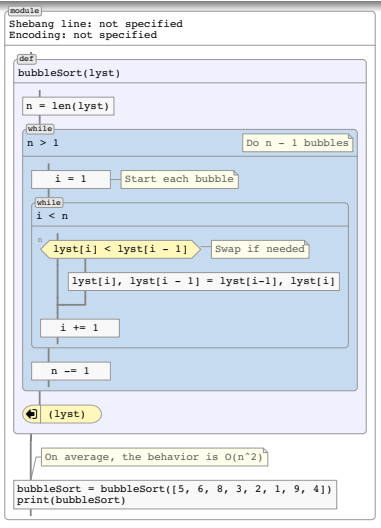
  Outer loop: Traverses the pairs in the given list, and on each pass it leaves one element out (bubbles it up)

  Inner loop: Compares the values of the engaged pairs and swaps the values if they are out of order

Example 9.4 (Script)

```
1  def bubbleSort(lyst):
2      n = len(lyst)
3      while n > 1:                        # Do n - 1 bubbles
4          i = 1                           # Start each bubble
5          while i < n:
6              if lyst[i] < lyst[i - 1]:  # Swap if needed
7                 lyst[i], lyst[i - 1] = lyst[i-1], lyst[i]
8              i += 1
9          n -= 1
10     return(lyst)
11
12 #On average, the behavior is O(n^2)
13 bubbleSort = bubbleSort([5, 6, 8, 3, 2, 1, 9, 4])
14 print(bubbleSort)
```

Example 9.4 (Control Flow)



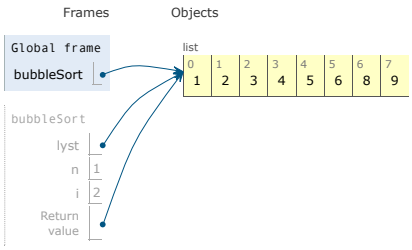
Example 9.4 (Program Execution)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  def bubbleSort(lyst):
2      n = len(lyst)
3      while n > 1:                        # Do n - 1 bubbles
4          i = 1                           # Start each bubble
5          while i < n:
6              if lyst[i] < lyst[i - 1]:  # Swap if needed
7                  lyst[i], lyst[i - 1] = lyst[i-1], lyst[i]
8              i += 1
9          n -= 1
10     return(lyst)
11
12 #On average, the behavior is O(n^2)
13 bubbleSort = bubbleSort([5, 6, 8, 3, 2, 1, 9, 4])
14 print(bubbleSort)
```

Print output (drag lower right corner to resize)

```
[1, 2, 3, 4, 5, 6, 8, 9]
```



Insertion Sort - Algorithm Design

- One input is needed: either an ordered or unordered list
- A nested loop is required:

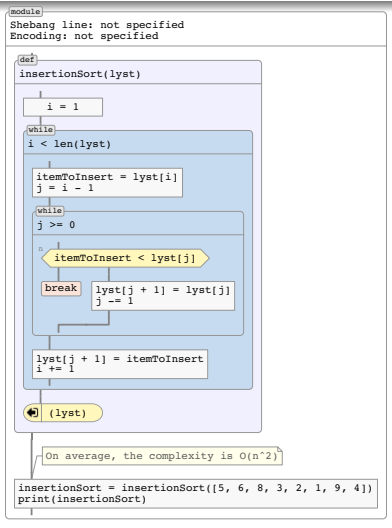
  Outer loop: Traverses the list from left to right starting with the second item

  Inner loop: Traverses the list from right to left starting with the engaged item. Inserts/updates the prior item with the value of the engaged item, if they are out of order

Example 9.5 (Script)

```
1  def insertionSort(lyst):
2      i = 1
3      while i < len(lyst):
4          itemToInsert = lyst[i]
5          j = i - 1
6          while j >= 0:
7              if itemToInsert < lyst[j]:
8                  lyst[j + 1] = lyst[j]
9                  j -= 1
10             else:
11                 break
12         lyst[j + 1] = itemToInsert
13         i += 1
14     return(lost)
15
16 #On average, the complexity is O(n^2)
17 insertionSort = insertionSort([5, 6, 8, 3, 2, 1, 9, 4])
18 print(insertionSort)
```

Example 9.5 (Control Flow)



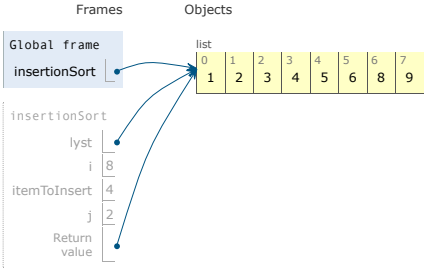
Example 9.5 (Program Execution)

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  def insertionSort(lyst):
2      i = 1
3      while i < len(lyst):
4          itemToInsert = lyst[i]
5          j = i - 1
6          while j >= 0:
7              if itemToInsert < lyst[j]:
8                  lyst[j + 1] = lyst[j]
9                  j -= 1
10             else:
11                 break
12         lyst[j + 1] = itemToInsert
13         i += 1
14     return(lyst)
15
16 #On average, the complexity is O(n^2)
17 insertionSort = insertionSort([5, 6, 8, 3, 2, 1, 9, 4])
18 print(insertionSort)
```

Print output (drag lower right corner to resize)

```
[1, 2, 3, 4, 5, 6, 8, 9]
```

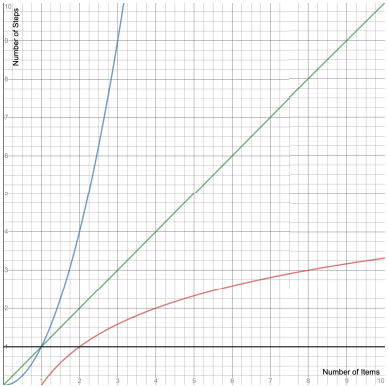


Algorithm Performance

Table 9.1: Types of Performance

Algorithm	Best Case	Average Case	Worst Case
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(log_2 n)$	$O(log_2 n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(N^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Figure 9.1: Big-O Computational Complexity Functions



References

- Lambert, K. A. (2010). Searching, sorting, and complexity analysis. In *Fundamentals of Python: From first programs through data structures* (pp. 431-459). Boston, MA: Cengage Learning

- Python Software Foundation. (2020). Python 3.7.8 documentation. Retrieved from `https://docs.python.org/3.7/index.html`.